**Università degli Studi di Pisa**

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Computer Engineering

Tesi di laurea magistrale

# Performance improvements on the P4 Software Switch

Candidato:
**Yuri Iozzelli**

Relatori:
**Prof. Luigi Rizzo**
**Prof. Giuseppe Lettieri**

**Anno Accademico 2015-2016**

**Abstract**

The Programming Protocol-Independent Packet Processors (P4) is a domain-specific language designed to allow programming of packet forwarding data-planes. It is used within a Software-Defined Networking (SDN) architecture, and it is meant to replace the Openflow protocol, being more flexible and powerful.

The reference software implementation of a P4 switch is currently used mainly to test the correctness of P4 programs, before deploying them on hardware switches. The software switch has poor performance in terms of throughput, and this prevent it from being used in many different scenarios, like network protocol experimentation and Virtual Machine interconnection.

The aim of this work is to analyze the bottlenecks of this implementation and improve the performance, trying to reach a throughput with an order of magnitude of 1 Mpps. Particular care has been used in making all the modifications as compatible as possible with the existing code base, in order to favour adoption of the improvements in existing and future applications.

# Contents

# Chapter 1

# Introduction

## 1.1 What is P4

Programming Protocol-Independent Packet Processors (in short, P4) is a domain-specific programming language designed to allow programming of packet forwarding dataplanes [1].

It is an open-source language maintained by a non-profit organization, the P4 Language consortium.

### 1.1.1 Design goals

The main design goals of the language are the following:

- **target independence:**

  P4 programs are designed to be implementation-independent, meaning they can be compiled against many different types of machines such as general-purpose CPUs, FPGAs, system(s)-on-chip, network processors, and ASICs. These are known as P4 targets, and each target must be provided along with a compiler that maps the P4 source code into a target switch model.

- **protocol independence:**

  P4 is designed to be protocol-independent, meaning that the language has no native support even for common protocols such as IP, Ethernet, TCP, etc... Instead, the P4 programmer describes the header formats and field names of the required protocols in the program, which are in turn interpreted and processed by the compiled program and target device.

- **reconfigurability:**

  Protocol independence and the abstract language model allow for reconfigurability – P4 targets should be able to change the way they process packets (perhaps multiple times) after they are deployed. This capability is traditionally associated with forwarding planes built on general-purpose CPUs or network processors, but the goal of P4 is to allow it also on fixed function ASICs.

### 1.1.2 Relation with Openflow

Both Openflow and P4 are used in the context of Software Defined Networking (SDN). In SDN, the contol plane is physically separated from the forwarding plane. This gives operators programmatic control over their networks.

Openflow is a common, open, vendor-agnostic interface which enable a control plane to control forwarding devices from different hardware and software vendors.

The original P4 paper [2] highlight the following problem with Openflow:

> The OpenFlow interface started simple, with the abstraction of a single table of rules that could match packets on a dozen header fields (e.g., MAC addresses, IP addresses, protocol, TCP/UDP port numbers, etc.). Over the past five years, the specification has grown increasingly more complicated (see tbl. 1.1), with many more header fields and multiple stages of rule tables, to allow switches to expose more of their capabilities to the controller. The

> proliferation of new header fields shows no signs of stopping [...] Rather than repeatedly extending the OpenFlow specification, we argue that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to leverage these capabilities through a common, open interface (i.e., a new "OpenFlow 2.0" API) [...]

Table 1.1: Fields recognized by the OpenFlow standard

| Version | Date | Header Fields |
|---|---|---|
| OF 1.0 | Dec 2009 | 12 fields (Ethernet TCP/IPv4) |
| OF 1.1 | Feb 2011 | 15 fields (MPLS, inter-table metadata) |
| OF 1.2 | Dec 2011 | 36 fields (ARP, ICMP, IPv6,etc) |
| OF 1.3 | Jun 2012 | 40 fields |
| OF 1.4 | Oct 2013 | 41 fields |

P4 has been design with the intention of being an improved version of the Openflow concepts, with the ability to define custom headers and tables, as well as explicitly programming the control flow of the switching logic (see fig. 1.1).

### 1.1.3 Match-action processing

Fundamental to P4 is the concept of **match-action pipelines**.

Conceptually, forwarding network packets or frames can be broken down into a series of table lookups and corresponding header manipulations. In P4 these manipulations are known as **actions** and generally consist of things such as copying byte fields from one location to another based on the lookup results on learned forwarding state.

P4 addresses only the data plane of a packet forwarding device, it does not specify the control plane nor any exact protocol for communicating state

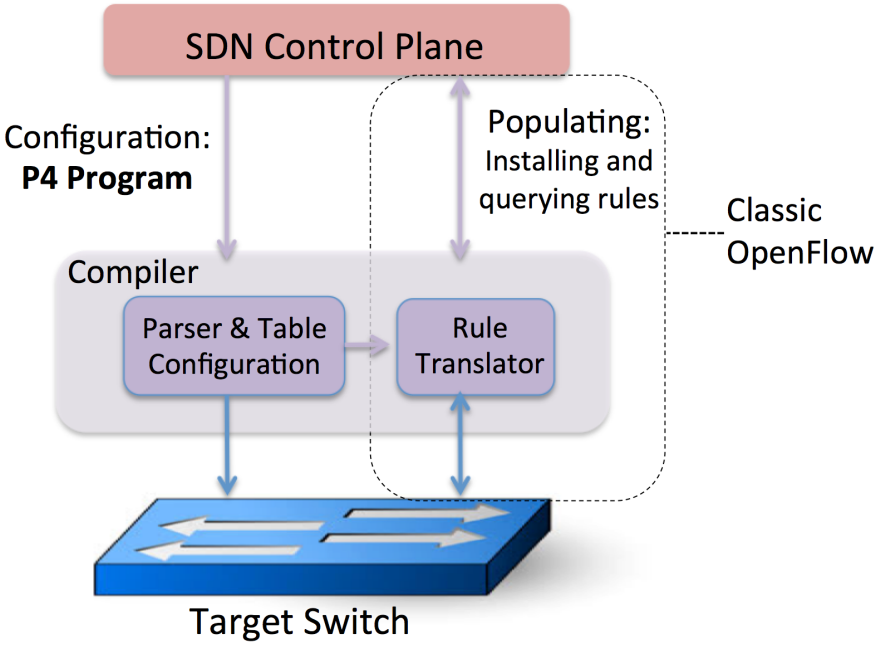


Figure 1.1: Configuration of a switch using P4

between the control and data planes. Instead, P4 uses the concept of tables to represent forwarding plane state.

An interface between the control plane and the various P4 tables must be provided to allow the control plane to inject/modify state in the program.

### 1.1.4 Language components

A P4 program is composed by the following fundamental elements:

- **Headers:** Header definitions describe packet formats and provide names for the fields within the packet. The language allows customized header names and fields of arbitrary length, although many header definitions use widely-known protocol names and fields widths. For example, an 802.3 ethernet header definition might be called “ethernet” and consist of the a 48-bit field named “dest” followed by a 48-bit “src” field, followed

by a 16-bit "type" field. The names in a header definition are used later in the P4 program to reference these fields.

- **Parsers:**

  The P4 parser is a finite state machine that walks an incoming byte-stream and extracts headers based on the programmed parse graph. A simple example would be a parser that extracts the ethernet source and destination and type fields, then performs a further extraction based on the value in the type field (common values might be ipv4, ipv6, or MPLS).

- **Tables:**

  P4 tables contain the state used to forward packets. Tables are composed of lookup keys and a corresponding set of actions and their parameters. A trivial example might be to store a set of destination MAC addresses as the lookup keys, and the corresponding action could set the output port on the device, and/or increment a counter. Tables and their associated actions are almost always chained together in sequence to realize the full packet forwarding logic, although in the abstract it is possible to build a single table that includes all the lookup key information and the full output action set.

- **Actions:**

  Actions in P4 describe packet field and metadata manipulations. In P4 context, metadata is information about a packet that is not directly derived from the parser, such as the input interface that the frame arrived on. English descriptions of an example action might be "decrement the IPv4 TTL field by one" or "copy the MAC address from the output port table into the outgoing packet header". P4 defines both standard metadata that must be provided by all targets as well as target-specific metadata, which is provided by the author of specific targets.

- **Control Flow:**

  The control flow in P4 determines the relative sequence of tables, and

allows for conditional execution of tables based on if/then/else constructions.

## 1.2 The behavioral model framework

P4 comes with a software reference implementation called **behavioral model**.

The first version of the behavioral model was written in C, and generated C code based on the P4 program logic. Because the code generation made the development process slow and difficult to follow, a new version called **bmv2** has been developed in C++11 [3].

The prominent feature of this version is that the C++ code of the switch is independent of the P4 program, which is fed to the switch at runtime. This allows the switch to be compiled only once, and even to swap the P4 program at runtime.

The bmv2 repository is written in a modular fashion and easily allow (and encourages) to write different targets with different features (hence the *framework* denomination used in this work).

The main goal of bmv2 is to allow vendors of hardware P4 switches to model their target and reproduce its behavior with different P4 programs.

## 1.3 Thesis work goal

The goal of this work is to improve the performance of the behavioral model framework in order to allow targets to achieve better throughput.

The main goal of the bmv2 repository is not performance, but correctness with respect to the specification of the P4 language. That said, a fast P4 switch implementation can be very useful in many scenarios, from the interconnection of Virtual Machines to the experimentation of new network protocols.

The current performance is low (400 Kpps for the simplest P4 program), but the code can be improved in many areas.

In particular the packet I/O subsystem is a good candidate to start working for improvements: In this work the **netmap** [4] framework will be used to provide a fast I/O layer to all targets.

A faster I/O subsystem will expose further bottlenecks, like the queuing mechanism used to connect different stages in the target pipeline (thus enabling a more efficient concurrent execution), and the memory allocation of the packet data structures.

In the following chapters we will discuss the cost associated with each subsystem operation, and provide a better implementation for some of them.

In doing this work, we tried to stick to the following principles:

- **Don't break existing target code**, unless there is a compelling reason to do so: this minimizes the effort to integrate the performance benefits to existing targets.
- **Minimize the modifications to existing framework files**: this allows for an easier rebase to future versions of the repository.

# Chapter 2

# Framework overview

In this chapter we will describe in more detail the structure of the bmv2 framework.

## 2.1 Terminology

First, it is better to formalize a few concepts. We already informally introduced some in the former chapter, and others are new:

- the **bmv2 framework** is the set of source files and libraries that make the reference software implementation of a switch capable of executing programs that follow the P4 specification.
- A **target** is an implementation of a P4 switch. A target built with the *bmv2 framework* is a **bmv2 target**.
- A **program**, or more precisely a **P4 program**, is a set of header definitions and match-action tables defined with the P4 language.
- A **parser**, in the context of a *target*, is a function that, given the current *program* and a newly arrived packet, extract the header information stored in the packet so that it can be accessed by the *target* and the *program*.

- A **deparser**, in the context of a *target*, is a function that, given the current *program* and a processed packet whose processing has ended, reconstruct the packet with the modified headers in front of the payload.
- A **control**, in the context of a *target*, is a function that, given the current *program* and a partially processed packet, apply to the packet the set of action defined in the corresponding *control* statement in the *program*.
- A **pipeline**, in the context of a *target*, is a set of parallel threads of execution, every one of which execute a part of the total processing that a packet must be subject to (which include receiving, parsing, one or more control stages, deparsing, and sending).
- A **queue**, in the context of a *target*, is both a buffer and a synchronization mechanism which connect two stages of a *target pipeline*.

In general, a target could support any number of **controls** defined in the P4 program, but standard practice in existing P4 programs makes use of a forwarding model consisting of a fixed pipeline with an **ingress control** and an **egress control**, so we will always refer to this kind of architecture (see fig. 2.1).

## 2.2 A closer look on bmv2

The repository contains three targets:

- **simple_router:**

  The simplest of the three. It has basic functionality and it is simple to understand

- **l2_switch:**

  Similar to the first, but includes a packet replication engine, to support multicast

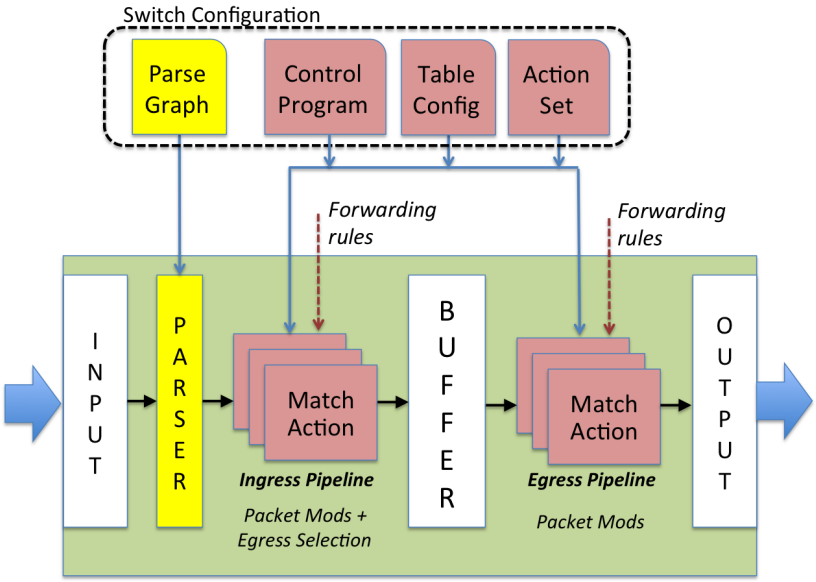


Figure 2.1: The standard forwarding model

- **simple_switch:**

  The standard P4 target, used to test and showcase all P4 features. It has, among the other features, a packet replication engine, a learning engine and support for priority queuing

There are many components provided by the framework. The following sections highlight the more critical for this work.

### 2.2.1 Packet I/O subsystem

This subsystem is responsible for fetching data packets from the attached network interfaces (real or virtual), and feed them to the switching logic, as well as pushing the processed data packets to the selected destination interface. There are currently many backends available (called **device managers**):

- a **pcap device manager**, which receives and sends packets on a network interface
- a **pcap file device manager**, which receives and sends packets using pcap files
- a **socket device manager**, which receives and sends packets on a socket

### 2.2.2 Packet processing subsystem

This subsystem is responsible for interpreting the P4 program control flow, and applying the required actions to each packet.

It does not use the program source directly, but a json representation of the parsed program produced by a tool named *p4c-bm*.

This subsystem behavior resembles that of an interpreter of a programming language, traversing all the control flow nodes and executing the actions it encounters.

### 2.2.3 Packet state representation subsystem

This subsystem is composed by a set of classes which hold the current packet payload and the parsed state of the program, composed of a series of header instances as defined by the P4 program. The header instances includes the metadata headers defined in the program.

The representation is split in 2 main classes: `packet` and `phv`. Every `packet` instance initially contains the original raw packet data, that is modified accordingly to the modifications made in the packet processing upon deparsing. Every `packet` is associated with a `phv`, which contains a representation of the header definitions of the P4 programs. The `phv` is what gives "structure" to the raw packet.

This distinction is important, because in the original code `phv` objects are recycled using a pool: this is because their construction is very expensive. `packet` objects, on the other hand, are quite cheap to create and destroy.

### 2.2.4 Communication interface with the control plane

A P4 switch starts with empty tables, just like an *Openflow* switch: it is a duty of the control plane to fill them with the right values.

The P4 specification does not specify how this communication has to be performed for P4 switches.

A target switch built with the bmv2 framework uses a *rpc server* that listens on a socket waiting for commands to fill the tables.

The commands are standard and common among all targets, and a simple CLI interface is provided with the framework for filling the tables with entries.

## 2.3 The simple_router target

In order to ease the following discussions on targets architecture, we will make use of a pseudo-code representation of the structure of target code.

In the bmv2 framework, every target has the same skeleton structure, which consist of:

- a `receive()` callback, invoked by the *device manager* for every incoming packet.
- A `new_packet()` function, called in the `receive()` callback. It gets the raw packet data as argument and it returns a `packet` object that can be used by the other functions
- one or more pipeline stages, each in its own thread, which perform one or more steps of the packet processing.

- A call to a `parse()` function, with a packet as argument, called in the `receive()` callback or in a pipeline stage.
- A call to a `ingress_control()` function, with a packet as argument, called in the `receive()` callback or in a pipeline stage.
- A call to a `egress_control()` function, with a packet as argument, called in the `receive()` callback or in a pipeline stage.
- A call to a `send()` function, with a packet as argument, called in the `receive()` callback or in a pipeline stage.
- One or more `queue` objects, with a `push()` and a `pop()` methods. The `push()` method blocks if the queue reached its maximum capacity, and the `pop()` method blocks if the queue is empty.

The **simple_router** target—which is the target we will refer to in most of this work—can be described by the (very simplified) following pseudo-code :

```
receive(raw_packet) {
    packet = new_packet(raw_packet);
    input_queue.push(packet);
}

process_thread() {
    while(true) {
        packet = input_queue.pop();
        parse(packet);
        ingress_control(packet);
        egress_control(packet);
        deparse(packet);
        output_queue.push(packet);
    }
}
```

```
output_thread() {
    while(true) {
        packet = output_queue.pop();
        if (to_send(packet)) {
            send(packet);
        }
    }
}
```

# Chapter 3

# Performance profiling

In this chapter we will analyze each operation described in sec. 2.3 in order to find the bottlenecks of the system.

## 3.1 Units of measurement

In order to measure the **throughput** of the whole target and of the single operations, we will use two main performance metrics:

- **packets per second (pps)**: this unit represent the throughput in terms of packets processed in the unit of time. We will use packets and not bits, because all the operations operate on single packets, and the payload is not much relevant on the cost of the operations (although it influences them, of course). For this reason—unless otherwise stated—we will use small packets for all the performance tests (~64 bytes).
- **seconds per packet (s/pkt)**: this unit is the reciprocal of the above one, and it may be more appropriate to reason in term of **s/pkt** when considering the cost of single operations, and the gain of using one optimization.

In sec. 4.5 we will also make some considerations about **latency** experienced by packets processed by the target. It will be measured in **seconds**.

All the above units will be used with *International System of Units (SI)* prefixes, whenever appropriate.

## 3.2 Test environment

All the numeric results of this work have been computed on a personal computer with the following specifications:

- **Operating System**: Arch Linux (linux kernel version 4.6.2)
- **CPU**: Intel Core i5-4690 CPU @ 3.50GHz (Quad-Core)
- **RAM**: 8 GB DDR3 1600 MHz

The target switch is configured with two open ports: one is attached to a packet generator, and another one with a packet receiver.

For both the generator and the receiver the program `pkt-gen` will be used. `pkt-gen` is a fast packet generator which uses netmap, and hence it is the perfect choice for our measurements.

On the testing machine, the generator can produce up to 46 Mpps with 64-Byte sized packets, if connected directly with the receiver via netmap pipes or patched veth drivers, and up to 3 Mpps with unpatched network drivers (see [4] for more information on netmap).

## 3.3 P4 test programs

We will perform the measurements with two p4 programs:

- **simple_router**: this program is shipped with the repository and it is the classic P4 example. It represents an L3 switch which chooses the next IP hop based on the source mac address, and then overwrite it with a mac address based on the exit port. It also updates the IP checksum field. Although it is a simple program, it is computational intensive compared to the basic_l2.

- **basic_l2**: this program has been written for the purpose of this work, and represents the simplest P4 program possible: it is an L2 switch that forward to the output port based on the destination MAC address. It does not modify the original packet. For its simplicity it is the perfect test to use to find inefficiencies in the framework architecture.

## 3.4 Methodology

In order to find the relative share of time spent on each operation, we will write a simplified target with only one thread and all the operations called in the `receive()` callback. We will start with an empty `receive()` in order to get the time spent in the device manager polling the interface, and then incrementally adding functionality and annotating the total time for each step. This way we can derive an estimation for the time spent in each operation. The following pseudo-code represent the architecture of this testing target:

```
receive(raw_packet) {                    //(1)
    packet = new_packet(raw_packet);     //(2)
    parse(packet);                       //(3)
    ingress_control(packet);             //(4)
    egress_control(packet);              //(5)
    deparse(packet);                     //(6)
    send(packet);                        //(7)
}
```

Once we have these basic measurements, we can try to estimate the cost of the synchronization between threads in the original `simple_router` target (that we remember has 3 concurrent thread in a pipeline, synchronized by two queues).

Because the first few steps of this process won't allow the receiver to receive packets, we will add a new thread to the target, specifically used to collect statistics on the throughput every few milliseconds.

## 3.5 Single operation costs

In tbl. 3.1 we can see the total time spent in the processing of one packet by progressively enabling the various operations in the testing target.

Table 3.1: Cumulative costs of the operations in the testing target

| | | simple router | basic l2 |
|---|---|---|---|
| 1 | device manager poll | 720 ns/pkt | 720 ns/pkt |
| 2 | packet creation/destruction | 925 ns/pkt | 925 ns/pkt |
| 3 | parsing | 1170 ns/pkt | 1030 ns/pkt |
| 4 | ingress control | 2090 ns/pkt | 1340 ns/pkt |
| 5 | egress control | 2530 ns/pkt | 1350 ns/pkt |
| 6 | deparsing | 2950 ns/pkt | 1450 ns/pkt |
| 7 | device manager send | 3800 ns/pkt | 2250 ns/pkt |

In tbl. 3.2 we can see the time spent for the single operations, derived from tbl. 3.1.

Table 3.2: Single costs of the operations in the testing target

| | simple router | basic l2 |
|---|---|---|
| device manager poll | 720 ns/pkt | 720 ns/pkt |
| packet creation/destruction | 205 ns/pkt | 205 ns/pkt |
| parsing | 245 ns/pkt | 105 ns/pkt |
| ingress control | 920 ns/pkt | 310 ns/pkt |
| egress control | 440 ns/pkt | 10 ns/pkt |
| deparsing | 420 ns/pkt | 100 ns/pkt |
| device manager send | 850 ns/pkt | 800 ns/pkt |

We can easily see from the tables that the most outstanding bottleneck is the device manager (and thus the packet I/O subsystem).

## 3.6 Simple router performance

Before addressing the packet I/O bottleneck, we will compute the overall performance of the simple_router target, as shown in tbl. 3.3.

Table 3.3: Overall performance of the simple_router target

| | simple router | basic l2 |
|---|---|---|
| simple_router target | 4920 ns/pkt | 1400 ns/pkt |

Surprisingly—despite using three threads—the simple_router has only a slightly better performance than the testing single-threaded target for the `basic_l2` program, and a **worse** performance for the `simple_router` program.

This means that, in addition to the packet I/O subsystem, we will need to design a better queuing and synchronization mechanism between pipeline stages. In sec. 4.2 we will explain in detail why the current mechanism is so slow, and present a much better alternative.

# Chapter 4

# Performance improvements

In this chapter we will discuss the improvements made to the packet I/O subsystem, the queueing mechanism, the thread pipeline and the memory allocations.

We consider the above costs to be **overhead** costs. The costs of parsing, deparsing and ingress/egress controls are considered **processing** costs.

In this work we will focus on reducing these overhead costs, because there is a clear margin of improvement in those areas.

Since the overhead costs increase both in percentage and in absolute value with the P4 program complexity, we think that it is more compelling to address those first, otherwise the great efforts needed to improve the processing costs may be made useless by the overhead bottlenecks.

Fig. 4.1 shows the overhead and processing costs for the simple_router and basic_l2 targets. The overhead costs are a significant fraction of the total.

Anyway, a better pipeline architecture will also reduce the processing costs.

## 4.1 Packet I/O

As we discovered in sec. 3.5, the main bottleneck is represented by the packet I/O subsystem.

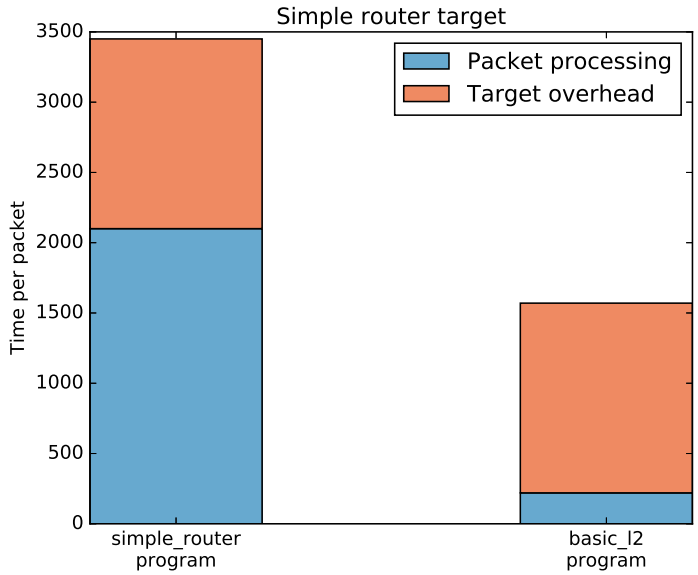


Figure 4.1: Processing and overhead times for the simple_router target

The class responsible for managing the dispatch and the reception of packets is the **device manager** (see sec. 2.2.1).

In order to achieve better performance we are going to implement a device manager which makes use of **netmap** [4] a framework specifically designed to enable fast packet I/O from userspace.

### 4.1.1 Performance evaluation

We will now compute the performance of the netmap device manager by using the testing target. This time we will skip all operations except the device manager poll, the packet creation, and the device manager send :

```
receive(raw_packet) {                    //(1)
    packet = new_packet(raw_packet);     //(2)
    send(packet);                        //(3)
}
```

The cumulative result is shown in tbl. 4.1.

Table 4.1: pcap and netmap device manager cumulative costs comparison

| | | pcap | netmap | netmap (lazy send) |
|---|---|---|---|---|
| 1 | device manager poll | 720 ns/pkt | 22 ns/pkt | 22 ns/pkt |
| 2 | packet creation | 930 ns/pkt | 230 ns/pkt | 230 ns/pkt |
| 3 | device manager send | 1700 ns/pkt | 780 ns/pkt | 320 ns/pkt |

As usual, we derive the single operation costs by subtracting each row from the above one (see tbl. 4.2).

Table 4.2: pcap and netmap device manager single operation costs comparison

| | pcap | netmap |
|---|---|---|
| device manager poll | 720 ns/pkt | 22 ns/pkt |
| packet creation | 210 ns/pkt | 208 ns/pkt |
| device manager send | 770 ns/pkt | 550 ns/pkt |

As we can see, using netmap drastically reduce the time it takes to receive packets, but the sending is not much better. In order to improve it, a short digression on how netmap achieve a faster throughput is needed

### 4.1.2 About netmap speed-up

Netmap achieve most of its speed compared to other solutions by batching system calls, thus splitting the cost between a big number of packets.

The `pkt-gen` script used in this work reports the average batch size for both the sender and the receiver ends. In tbl. 4.3 we can see the difference in batch size at the receiver end between the pcap version and the netmap version of the target: the netmap target propagates the batch to the receiver, while the pcap one does not.

Table 4.3: avg batch size measured by the receiver for pcap and netmap device managers

| | pcap | netmap |
|---|---|---|
| average batch size | 22 pkts | 512 pkts |

The pcap device manager is not able to exploit the batched arrival of packets, and does not propagate the batching to the receiving process.

While the `poll()` operation in our netmap device manager exploits the packet batching, the current implementation of the `send()` does not. It is indeed more complicated to use batching while sending, because it is not possible to know in advance if the current packet will be the last of the batch, or if new packets are arriving: if we perform the sync operation with the netmap driver only when a certain number of packet is queued (e.g. 512), the last packets of the batch may remain an indefinite amount of time in the queue (for example if a batch of 513 packets arrive, the last one will be kept in queue until 511 more packets arrive).

Since this is not acceptable, a possible solution may be to periodically flush the output queue of the device manager even if the batch is not completed.

Fortunately we don't need another thread to perform this task, because netmap can be configured so that the `poll()` system call synchronize both the read and the write rings of the netmap device. We just need to add a timer to the `poll()`, so that even if no packet arrived, we are guaranteed that periodically the output queue is flushed.

Doing this optimization we improve the sending cost to that of tbl. 4.4.

Table 4.4: pcap and netmap device manager single operation costs comparison (improved)

| | pcap | netmap (improved) |
|---|---|---|
| device manager poll | 720 ns/pkt | 22 ns/pkt |
| packet creation | 210 ns/pkt | 208 ns/pkt |

| | pcap | netmap (improved) |
|---|---|---|
| device manager send | 770 ns/pkt | 90 ns/pkt |

### 4.1.3 Simple router performance

We will now repeat the overall packet time test of the simple_router target, now with netmap support (see tbl. 4.5).

Table 4.5: Overall performance of the simple_router target with netmap device manager

| | simple router | basic l2 |
|---|---|---|
| simple_router target (netmap) | 4700 ns/pkt | 1900 ns/pkt |

Surprisingly, the packet time for the simple_router target has only a slight gain, while the basic_l2 target performs **worse** than the version without netmap. This is caused by the poor performance of the implementation of the queue between pipeline stages: If We recall the structure of the simple_router target , we realize that now the first and third stage should be fast, but the middle stage remain slow, and this cause the input queue to be always full, and the ouptput queue to be always empty. Thus— because the queue is implemented with locks and condition variables—a notification is sent and a thread is locked/unlocked for **each** packet.

The overhead of this mechanism is not bearable, and will be addressed in the next section.

## 4.2 Queuing mechanism

Before trying to improve the design of the queue, we should try to understand what is wrong with the current one.

We start by writing a testing program with two threads connected by a queue. The queue will contain integers and the threads will push and pop the queue at the maximum speed possible. The result is in tbl. 4.6 (this time we measure the average time to push and pop a single item).

Table 4.6: Locking queue time to transfer a single item via push/pop between two threads

| | locking queue |
|---|---|
| testing queue program | 250 ns/item |

The queue takes way too long to transfer packets between stages, so we will design a better one.

### 4.2.1 Current queue design

The current queue makes use of a mutex and two condition variables (for the empty queue and the full queue conditions). It can be described by the following pseudo-code:

```
template<typename T>
class Queue {
 public:
  void push(T item) {
    lock(_mutex);
    while (_deque.size() == max_size) {
      cond_not_full.wait();
    }
    _deque.push_front(T);
    unlock(_mutex);
    cond_not_empty.notify();
  }
  T pop() {
    lock(_mutex);
```

```
    while (_deque.size() == 0) {
      cond_not_full.wait();
    }
    T item = _deque.pop_back();
    unlock(_mutex);
    cond_not_full.notify();
  }
 private:
  condition_variable cond_not_full;
  condition_variable cond_not_empty;
  mutex _mutex;
  deque<T> _deque;
};
```

There are two problems with this design:

- Every operation tries to acquire a lock on the same mutex, thus creating an high contention between threads. Even without contention, acquiring a lock is somewhat expensive if we measure things in the order of tens of nanoseconds.
- Every operations notify the other thread's condition variable: The notify itself is expensive, and we could think of firing it only if the queue if really empty or full, and not every time. In any case, the system will probably end up in a situation in which a queue is almost always empty or almost always full, so the queue will fire one time every two operations anyway.

### 4.2.2 Lockless queue idea

Since the push operation in principle operates on the front of the queue, and the pop operation operates on the back of the queue, it should be possible to relax the constraint to lock the global mutex for every operation.

We will restrict the queue usage to the **single producer, single consumer** case: while the original queue can be used by any number of threads, the queue that we will design will only work when exactly one thread uses the `push()` operation, and exactly one thread uses the `pop()` operation.

This ensure that only the **producer** will update the front of the queue, and only the **consumer** will update the back of the queue. There is no need of protecting those variables with a mutex.

### 4.2.3 A basic lockless queue

To build a basic lockless queue we can use a fixed-size array (whose size is set to the capacity of the queue) and two indexes: one indicates the next available slot for the producer, and the other the next ready slot for the consumer.

The array is used as a ring buffer, so the indexes wrap around the array when they reach the end.

The consumer index can never pass over the producer index. When the two are the same the queue is empty, and when the producer index is right behind the consumer index, the queue is full.

A simplified implementation could be:

```
template<typename T>
class LocklessQueue {
 public:
  void push(T item) {
    while (wrap(cons_idx+N-1) == prod_idx) {
      cond_not_full.wait();
    }
    ring[prod_idx] = item;
    prod_idx = wrap(prod_idx+1);
    cond_not_empty.notify();
  }
  T pop() {
```

```
    while (prod_idx == cons_idx) {
      cond_not_empty.wait();
    }
    T ret = ring[cons];
    cons_idx = wrap(cons_idx+1);
    cond_not_full.notify();
    return ret;
  }
 private:
  int wrap(int idx) {
    return (idx%N+N)%N;
  }

  condition_variable cond_not_full;
  condition_variable cond_not_empty;
  T[N] ring;
  atomic_int cons_idx{0};
  atomic_int prod_idx{0};
};
```

See figs. 4.2, 4.3, 4.4 for a visual representation of different queue states.

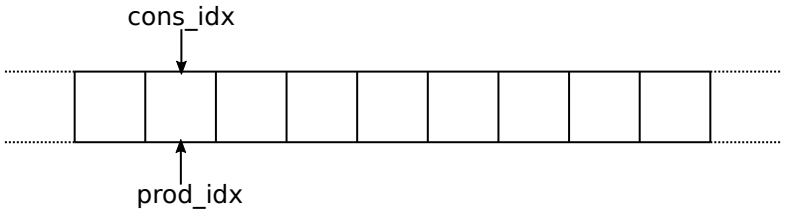


Figure 4.2: Visual representation of an empty queue state

#### 4.2.3.1 Atomicity and memory barriers

The `cons` and `prod` indexes must be atomic values, and each time their value is read or written, an appropriate memory barrier must be issued. Otherwise

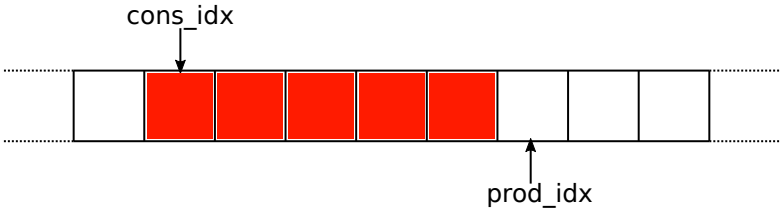


Figure 4.3: Visual representation of an intermediate queue state

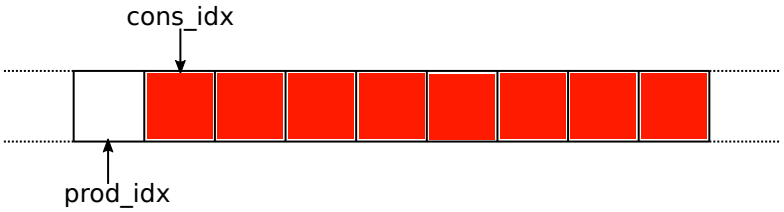


Figure 4.4: Visual representation of a full queue state

concurrent cores in the cpu may not see the change of the variable value as it occurs.

In this work we make use of the atomic types available in the C++11 standard (in the `<atomic>` header): they wrap the normal basic types and ensure that appropriate instruction for the target architecture are generated to ensure the atomicity of each operation they are involved in. They also allow to specify the type of memory ordering that the programmer want to ensure for each operation.

#### 4.2.3.2 Performance evaluation

We ran the same performance test as tbl. 4.6, and we obtained the result in tbl. 4.7.

Table 4.7: Basic lockless queue time to transfer a single item via push/pop between two threads

| | basic lockless queue |
|---|---|
| testing queue program | 300 ns/item |

As we can see, this lockless queue performs slightly **worse** than the regular one, for the following reasons:

- **Atomic variables overhead:**

  Accessing atomic variables is of course slower than accessing regular ones (because a cpu core may be denied access to the bus if an atomic operation is ongoing, and cache may be invalidated), so it would be wise to manually cache reads (and even writes) whenever possible when handling atomic variables.

- **unnecessary notifications:**

  As in the locking queue case, it is not necessary to notify the condition for every operation, but only when the condition effectively changes. This is not trivial to do, because race conditions may arise and result in the loss of one notification, thus leading to a deadlock.

- **lack of hysteresis:**

  As in the locking queue case, if one of the threads is faster than the other the queue will be empty or full most of the time. Even if we notify only when needed, we will likely need to notify for each packet at the steady state. We would thus prefer to wait some time for the queue to empty (or full) a little before notifying the other thread.

### 4.2.4 A better lockless queue

We want to design a lockless queue that updates shared state as little as possible, suppress redundant notifications and implements some kind of hysteresis to avoid an always full or always empty state.

#### 4.2.4.1 Notification suppression

To solve the last two problems, we will make use of the idea of **notification events**, borrowed from the *Virtio network drivers* [5].

The idea is the following: both the producer and the consumer have, besides their current index, an event index. This index is used to tell the other thread when the current thread wants to be notified and woken up.

For example if the producer thread finds the queue full, he may want to be notified when the queue is half empty, and not when the queue is full except for a single slot (which will force the producer to go to sleep again).

Figs. 4.5, 4.6, 4.7 illustrate an example in which the producer wait on the full queue condition and sets a `prod_event` in order to be notified when the queue is partially empty, and not as soon as it is not full.

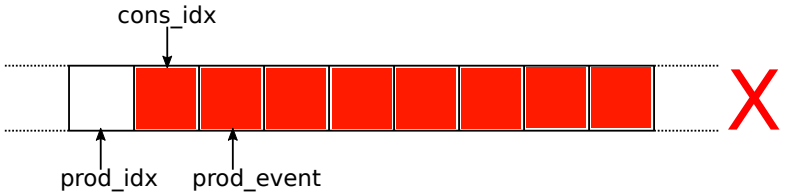


Figure 4.5: Producer finds full queue, sets prod_event. Consumer does not signal

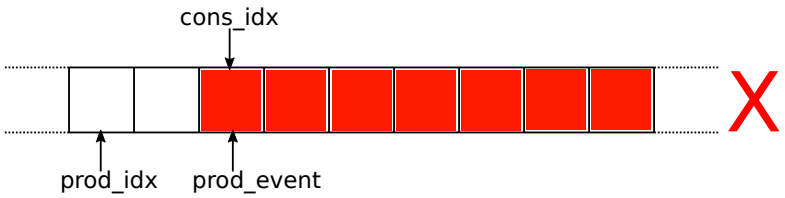


Figure 4.6: Producer still blocked. Consumer does not signal

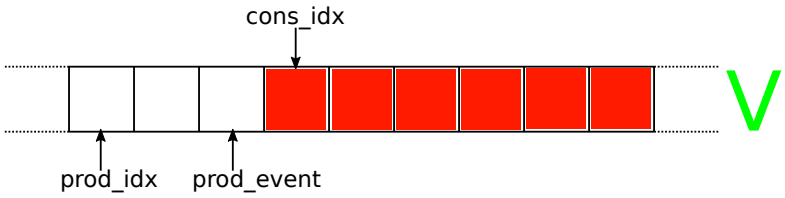


Figure 4.7: Consumer passed over prod_event and signals Producer

There are some constrains to this technique though: the consumer cannot reliably set an event that does not correspond to the current `prod_index`.

An example will easily show why:

1) the queue is empty and the consumer sets a `cons_event` to two packets ahead the current `prod_index`

2) One more packet arrive in the queue, then no packet arrive for the next 10 minutes

3) The last packet in the queue suffer a latency of 10 minutes, which is not acceptable.

For the producer this problem does not arise, because the consumer will always try to consume the queue as fast as it can, and the producer is guaranteed that the queue will reach the desired size in a finite amount of time.

Another issue with this technique is that there is a potential race between the advance of an index and the setting of an event. For example take this scenario:

1) The queue is full and the producer sets a `prod_event` two slots ahead the current `cons_index`.

2) In between the check that the queue is full and the setting of the `prod_event`, the consumer pops 4 items from the queue.

3) The consumer does not see that it passed over the `prod_event`, because the event has still its previous value, so the consumer will not fire the signal.

4) The producer will never wake up, and the entire program freezes.

The solution to this problem is to **double check** the wait condition after setting the event. The same scenario as before now becomes:

1) The queue is full and the producer sets a `prod_event` two slots ahead the current `cons_index`.

2) In between the check that the queue is full and the setting of the `prod_event`, the consumer pops 4 items from the queue.

3) The producer—**before** going to sleep—checks again that the queue is full.

4) Since the queue is not full anymore, the producer does not go to sleep, and can insert items right away.

#### 4.2.4.2 Reduce contention on shared state

The other problem with the basic queue is the access to the shared atomic variables: In order to increase concurrency, the threads should access shared state as little as possible. The *Intel Guide for developing Multithreaded Applications* [6] suggests (among others) the following expedients to noticeably increase performance in concurrent environments:

- Put variables mainly used by different threads on different cache lines to avoid **false sharing**:

  Each core in modern CPUs has a local cache. The memory system mus guarantee cache coherence. False sharing occurs when threads on different cores modify variables that resides on the same cache line. This invalidates the cache line and forces an update, which hurts performance (see fig. 4.8).

  In our case, the `cons_index` and the `cons_event` variables—updated by the consumer—should be on a different cache line than the `prod_index` and the `prod_event` variables—updated by the producer.

- Use **thread-local storage** to reduce synchronization:

  This expedient—which we will use further on about memory allocation—suggest to avoid the access of shared data if possible, instead relying on variables used only on the local thread. In our case, since we are sure only one thread uses the consumer side, and only one thread uses the producer side, we can simply use non-atomic variables local to the thread for storing the main indexes. The shared atomic indexes will only be updated if needed.

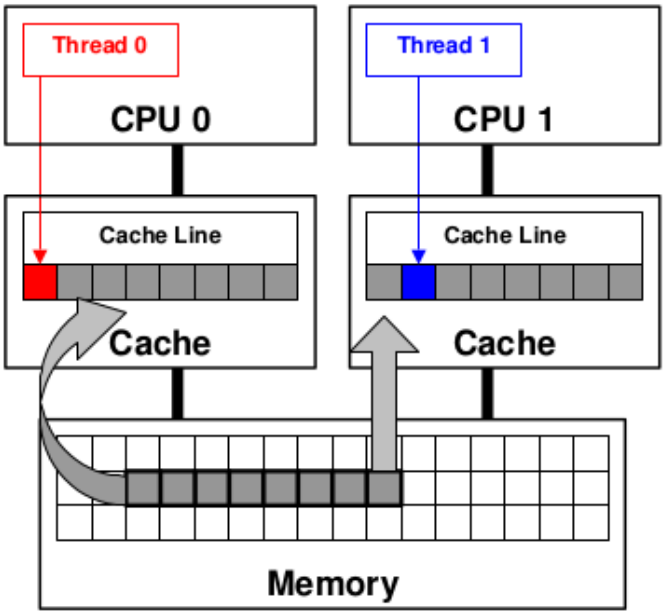


Figure 4.8: Concurrent writes from different threads on variables on the same cache line

The second point needs further explanation: doesn't the producer need to know the exact position of the `cons_index`, and vice-versa doesn't the consumer need to know the exact position of the `prod_index`?

Let's examine the situation, starting from the **producer**:

- The producer is **required** to update the shared `prod_index` before going to sleep, otherwise it may never get the chance to do it again if the consumer thinks the queue is empty. It is probably not **convenient** to update so rarely though, because it can affect concurrency. For this reason we added a `bool force` parameter to the `push()` method. If the parameter is set, the producer will update its index, regardless of the state of the queue. If the function that calls the method is aware that more packets are coming, it can set the parameter only for the last packet of the batch.
- The producer has **convenience** to update its local copy of the

`cons_index` if the queue seems full, because the cost of updating the index and possibly avoiding blocking the thread is lower than actually blocking the thread.

The situation for the consumer is almost symmetrical, but there is no need to explicitly convey batching information: The consumer know how much elements are in the queue (to be more correct, it knows how much elements the producer made available through the shared index), so it can pop them all with the synchronization cost of a single `pop()`. For this reason we added a new `pop()` overload that returns an array with all the elements found in the queue.

To better explain the whole system, let's see the following example (in the following figures the indexes shown above the queue are those visible from the consumer, and the ones shown below are those visible by the producer):

1) The queue starts empty (fig. 4.9).

2) he producer adds in sequence 4 elements, setting the `force` flag only for the last one: from the consumer point of view, this is equivalent to a single insertion of 4 elements (fig. 4.10).

3) The producer notifies the consumer, but before the latter wakes up it inserts 2 more packets without the `force` flag (fig. 4.11).

4) The consumer wakes up—notified by the producer at the fourth insertion—and pop all the elements it can see (4) from the queue (fig. 4.12).

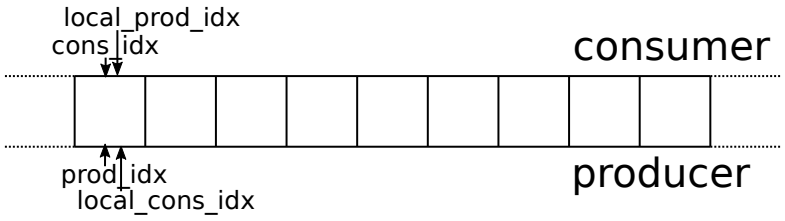


Figure 4.9: Empty queue

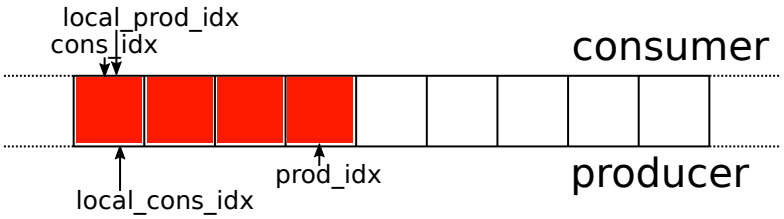


Figure 4.10: Producer adds 4 elements. Last push has `force` argument set. Consumer is notified

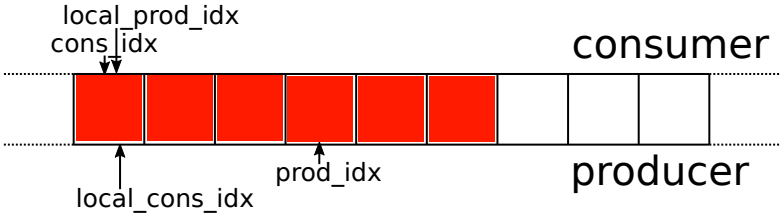


Figure 4.11: Producer adds 2 elements

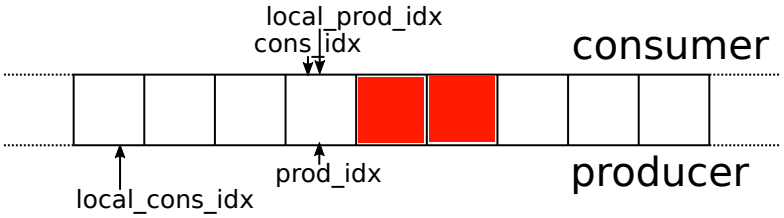


Figure 4.12: Consumer wakes up, updates local prod_index and consumes 4 elements

#### 4.2.4.3 Virtual queue ring

A minor but not negligible performance hit is given by the modulo operation needed every time an index is used.

To avoid this penalty we can can use a *virtual ring* of size $2^N$. The indexes used in the queue will be of type `uintN_t` (this limit the value of N to 8, 16, 32 or 64). In this way, when an index reaches the end of the virtual ring, it automatically wraps around without any operation needed.

The virtual ring will be mapped onto the real ring (of size $K$, which have to be a power of 2) by simply masking the upper part of the indexes.

Using this system we can also use the full size of the real ring, without having to leave an empty slot between `cons_index` and `prod_index`, because if $K < N$ if the two indexes are equal it unequivocally means that the queue is empty.

The size of the queue $S$ can still be other than a power of 2: the full queue condition will use $S$ and not $K$ (which will be the nearest power of two greater or equal than $S$) as the maximum vaule. If this is the case, of course the real ring will never be fully used.

#### 4.2.4.4 Performance evaluation

We ran the same performance test as tbl. 4.6 and tbl. 4.7, and we obtained the result in tbl. 4.8.

Table 4.8: Better lockless queue time to transfer a single item via push/pop between two threads

| | better lockless queue |
|---|---|
| testing queue program | 150 ns/item |

This queue is twice as fast as the basic version, but still not fast enough considering all the improvements.

The reason the performance is still not optimal is because, while the producer can benefit from setting the `prod_event` further ahead the `cons_index`, the consumer cannot do that (see sec. 4.2.4.1).

Tbl. 4.9 shows the number of **average notifications per packet (not/pkt)** sent from each thread for the three kind of queue we examined so far.

Table 4.9: Average notifications per packet for the three queue types

| | locking | lockless (basic) | lockless (better) |
|---|---|---|---|
| producer | 1 not/pkt | 1 not/pkt | 0.6 not/pkt |
| consumer | 1 not/pkt | 1 not/pkt | $1.6 \cdot 10^{-6}$ not/pkt |

We want to reduce the number of notifications that the producer sends to the client. They are proportional to the number of times the consumer wait on the empty queue condition.

We already know we cannot use the `cons_event` for this (like in the producer case), because it would introduce unbounded wait time. What we can do is explicitly wait for a bounded amount of time: If the consumer is going to block on the empty queue condition, it will first sleep for a given number of microsecond first. When it wakes up, it checks the condition again, and it goes to sleep only if the queue is still empty.

With the above optimization the number of notifications per packet sent by the producer drastically reduces (see tbl. 4.10).

Table 4.10: Better queue performance after the consumer sleep optimization

| | lockless (better) |
|---|---|
| producer | $1.2 \cdot 10^{-6}$ not/pkt |
| consumer | $8 \cdot 10^{-4}$ not/pkt |

We can now make the throughput measurement again and see that the notification suppression is indeed effective (see tbl. 4.11). The resulting time

is around 6 times less than the original locking queue.

Table 4.11: Better lockless queue time to transfer a single item via push/pop between two threads.

| | better lockless queue |
|---|---|
| testing queue program | 50 ns/item |

The overall throughput achieved by using this queue in the simple_router target is shown in tbl. 4.12.

Table 4.12: Overall performance of the simple_router target with netmap device manager and better lockless queue.

| | simple router | basic l2 |
|---|---|---|
| simple_router target (netmap+lockless) | 3300 ns/pkt | 1120 ns/pkt |

The improvement made to the queue result in only a modest improvement on the overall performance of the simple_router target. In order to unlock their full potential we need to improve on the other fronts too.

## 4.3 Better pipeline design

Now that our queue is efficient, we can think about changing the pipeline structure. The simple_router target structure is described by sec. 2.3, and we can notice looking at it that the load on the various threads is not well balanced. In tbl. 4.13 we show the single operation costs we already calculated, with the costs for packet I/O updated with netmap usage.

Table 4.13: Single costs of the operations in the testing target

| | simple router | basic l2 |
|---|---|---|
| device manager poll | 22 ns/pkt | 22 ns/pkt |
| packet creation/destruction | 205 ns/pkt | 205 ns/pkt |
| parsing | 245 ns/pkt | 105 ns/pkt |
| ingress control | 920 ns/pkt | 310 ns/pkt |
| egress control | 440 ns/pkt | 10 ns/pkt |
| deparsing | 420 ns/pkt | 100 ns/pkt |
| device manager send | 90 ns/pkt | 90 ns/pkt |

Grouping the costs for every thread we get the results in tbl. 4.14 (the packet creation/destruction costs have been split between `receive()` and `output_thread()`).

Table 4.14: Costs of the operations grouped by thread in the simple_router target

| | simple router | basic l2 |
|---|---|---|
| receive | 105 ns/pkt | 105 ns/pkt |
| process_thread | 1915 ns/pkt | 525 ns/pkt |
| output_thread | 192 ns/pkt | 192 ns/pkt |

As we can see, the load is poorly balanced among threads, even in the basic_l2 case, where the processing work is low.

We thus propose an alternative target structure, which we will call *fast_switch*. This target will have 4 threads instead of 3, the `receive()` callback will also do the packet parsing, an `ingress_thread()` will only do the ingress control, an `egress_thread()` will only do the egress control, and the `output_thread()` will also do the deparsing.

In tbl. 4.15 we can see the theoretical grouped cost for each thread.

Table 4.15: Costs of the operations grouped by thread in the simple_router target

| | simple router | basic l2 |
|---|---|---|
| receive | 350 ns/pkt | 210 ns/pkt |
| ingress_thread | 910 ns/pkt | 310 ns/pkt |
| egress_thread | 440 ns/pkt | 10 ns/pkt |
| output_thread | 602 ns/pkt | 292 ns/pkt |

The pseudo-code representation of the architecture is the following:

```
receive(raw_packet) {
    packet = new_packet(raw_packet);
    parse(packet);
    ingress_queue.push(packet);
}

ingress_thread() {
    while(true) {
        packet = ingress_queue.pop();
        ingress_control(packet);
        egress_queue.push(packet);
    }
}

ingress_thread() {
    while(true) {
        packet = egress_queue.pop();
        egress_control(packet);
        output_queue.push(packet);
    }
}
```

```
output_thread() {
    while(true) {
        packet = output_queue.pop();
        deparse(packet);
        if (to_send(packet)) {
            send(packet);
        }
    }
}
```

It would maybe be better to further split the `ingress_thread()` and `egress_thread()` threads, because its complexity grows with the complexity of the P4 program. There could be for example a thread for every action defined in the control.

This approach have two problems though:

- A practical one: The bmv2 framework is not flexible enough to expose the single actions to the target code, so a heavy modification to it would be needed.
- A theoretical one: Too many pipeline stages means more latency (see sec. 4.5 for more about this), and a large number of threads means more synchronization overhead. Also, if the target machine has less cores than the number of threads used, the performance gain will be negligible (or worse negative).

The fast_switch architecture is a good balance between thread load and pipeline size.

### 4.3.1 Performance evaluation

The overall throughput achieved by the fast_switch target is shown in tbl. 4.16.

Table 4.16: Overall performance of the fast_switch target.

| | simple router | basic l2 |
|---|---|---|
| fast_switch target | 1750 ns/pkt | 900 ns/pkt |

This is a significant improvement over the original simple_router target.

## 4.4 Memory allocations

Both in the simple_router and the fast_switch target every packet entering the system is allocated in the `receive()` callback and deallocated in the `output_thread()` thread. This is a problem because the standard `malloc` implementation uses global locks in order to protect the heap data structures from concurrent access.

This means that, since packets are created and destroyed all the time, there is a lot of contention on those locks. The real time wasted in these operations cannot be seen in tbl. 3.2, because those times are measured in a single-thread environment. Also, the penalty increases with the overall speed of the system.

### 4.4.1 Custom packet allocator

We are going to redefine the `new` and `delete` operators of the `Packet` class. To make things simpler, we will start by assuming a constant size of 2048 bytes for the packet buffer. This way the total size of a packet instance is fixed and known at compile time.

The redefined operators will manage heap memory independently from the system `malloc`, but will resort to it if more memory is needed.

They will never return memory to the system. This will not result in an unbound growth of memory, because the amount of packets present in the switch at any given time is limited by sum of the size of all the queues.

In order to reduce thread contention, every thread will have its own `thread_local` list of available memory chunks. In practice this means that the thread that creates packets will take chunks from its list, and the thread that destroys packets will release chunks to its list. The `thread_local` lists have a maximum size.

In addition to the local lists, a single global list—protected by a global mutex—exists.

When a local list is empty and a new chunk is requested, the thread will acquire the global mutex and fill its local list with the content of the global list. If the global list is empty too, the thread will resort to calling the standard `malloc`.

When a local list is full and a used chunk is released, the thread will acquire the global mutex and empty its local list into the global one.

This system amortize the cost of using the global lock by using it to group many insertion/deletions in one global operation.

An example simplified implementation is shown below:

```
static std::list<void*> global_freelist;
static std::mutex global_mutex;
static thread_local std::list<void*> local_freelist;
static const size_t max_local_size{1024};


static void* Packet::operator new(std::size_t sz) {
  (void)sz;

  if (local_freelist.size() == 0) {
    std::unique_lock<std::mutex> lock(global_mutex);
    if (global_freelist.size() > 0) {
      local_freelist.merge(global_freelist);
    }
    lock.unlock();
```

```
    if (local_freelist.size() == 0) {
      local_freelist.push_back(malloc(sizeof(Packet)));
    }
  }
  void* p = local_freelist.front();
  local_freelist.pop_front();
  return p;
}

static void Packet::operator delete(void* p) {
  local_freelist.push_back(p);
  if (local_freelist.size() > max_local_size) {
    std::unique_lock<std::mutex> lock(global_mutex);
    global_freelist.merge(local_freelist);
    lock.unlock();
  }
}
```

The real implementation uses a custom single-linked list which uses the same chunk of memory to store both the list pointers and the packet memory via a union (this avoids allocations/deallocations for the list structure).

#### 4.4.1.1 Performance evaluation

We now evaluate the performance of this improvement on the fast_router target (see tbl. 4.17).

Table 4.17: Overall performance of the fast_switch target with and without the custom allocator.

| | simple router | basic l2 |
|---|---|---|
| fast_switch target | 1750 ns/pkt | 900 ns/pkt |
| fast_switch target (custom allocator) | 1330 ns/pkt | 450 ns/pkt |

As expected, the already faster basic_l2 target gains a higher speedup because it had more contention on the malloc lock.

### 4.4.2 Tcmalloc

Tcmalloc (thread-caching malloc) is a malloc implementation, and is part of the gperftools suite, developed by Google. It has a permissive BSD-like licence.

It is optimized for high concurrency scenarios like the one in this work.

The basic idea is the same of the custom packet allocator we built in the previous section, but has many advantages over it:

- **It is not limited to chunks of a given size:**

  To accomplish this without incurring in fragmentation it distinguish "small objects" from "large objects". Small objects are subdivided in 86 classes of different sizes, each one with its own set of lists. The large objects are allocated in terms of 4k pages.

  We can thus use tcmalloc with arbitrary sized packets.

- **It is a transparent malloc replacement:**

  This means that in order to use it, we don't need to change a single line of code, but just link `libtcmalloc.so` to the executable. It is also possible to link it at runtime using the environmental variable `LD_PRELOAD`.

- **It applies to every allocation, not only packets:**

  While this is not a big improvement on performance now (because packet objects are the only ones that are continuously allocated/deallocated from different threads), it allows future versions of targets to scale well if new data structures are needed, without having to write an allocator for every one of them.

The only downside is that performance is slightly worse than the custom allocator one (see tbl. 4.18), and the total memory footprint of the switch slightly increases.

Table 4.18: Overall performance of the fast_switch target with the custom allocator and with tcmalloc.

| | simple router | basic l2 |
|---|---|---|
| fast_switch target (custom allocator) | 1330 ns/pkt | 450 ns/pkt |
| fast_switch target (tcmalloc) | 1450 ns/pkt | 520 ns/pkt |

But because one of the thesis work goal was to minimize modifications to the original code, we think that the pros overcome the cons.

## 4.5 Optimal queue size and latency

For now we didn't talk about the size of the queues used in the targets.

The default sizes in the simple_router target are 1024 for the input queue and 128 for the output queue. In the fast_switch target we kept these sizes and the extra queue is also 1024 in size.

Since all our experiments stress the targets to the maximum speed possible, at least one queue is full most of the time. This means that we can approximate the average number of packets in the system with the size of the queues.

If we increase the size of the queues then, from **Little's law**, we have an increase in latency. But if we decrease the size too much we increase the synchronization overhead and we end up with a worse throughput. The relation between queue size, throughput and latency thus is trivial to figure.

We can measure throughput and latency for different values of the queue size, and decide a good trade-off between the two.

First, as a base measurement, we can see the values for the original simple_router target and the original fast_switch target with the basic_l2 P4 program (tbl. 4.19).

Table 4.19: Throughput and latency measurements for the two targets .

| | throughput | average latency | max latency |
|---|---|---|---|
| simple_router target | 670 Kpps | 0.26 ms | 26 ms |
| fast_switch target | 1950 Kpps | 0.58 ms | 8.5 ms |

We can see that, even if the throughput of the fast_switch target is much higher than that of the simple_router target, average latency is worse.

This is because the fast_switch has an additional pipeline stage, and thus more packets in the system on average. Because of Little's law this means more latency. The maximum latency is smaller though, because the system is faster overall due to the reduced number of locks and contentions.

Focusing on the fast_switch target, we can try various configurations of queue size and see how this influences throughput and latency. For simplicity we will use the same size for all the three queues (tbl. 4.20).

Table 4.20: Throughput and latency measurements for different queue sizes on the fast_switch target .

| queue size | throughput | average latency | max latency |
|---|---|---|---|
| 1 pkt | 18.0 Kpps | 0.21 ms | 8.9 ms |
| 2 pkt | 38.0 Kpps | 0.17 ms | 9.0 ms |
| 4 pkt | 72.6 Kpps | 0.13 ms | 7.8 ms |
| 8 pkt | 145.5 Kpps | 0.10 ms | 4.7 ms |
| 16 pkt | 356.0 Kpps | 0.10 ms | 7.0 ms |
| 32 pkt | 760.1 Kpps | 0.092 ms | 7.3 ms |
| 64 pkt | 1441 Kpps | 0.093 ms | 7.8 ms |
| 128 pkt | 2243 Kpps | 0.11 ms | 9.0 ms |
| 512 pkt | 2350 Kpps | 0.28 ms | 7.6 ms |
| 1024 pkt | 1945 Kpps | 0.60 ms | 9.7 ms |
| 2048 pkt | 1730 Kpps | 1.20 ms | 15 ms |
| 4096 pkt | 1486 Kpps | 3.16 ms | 15 ms |

It is interesting to plot the queue size against the throughput and the latency (see figs. 4.13, 4.14).

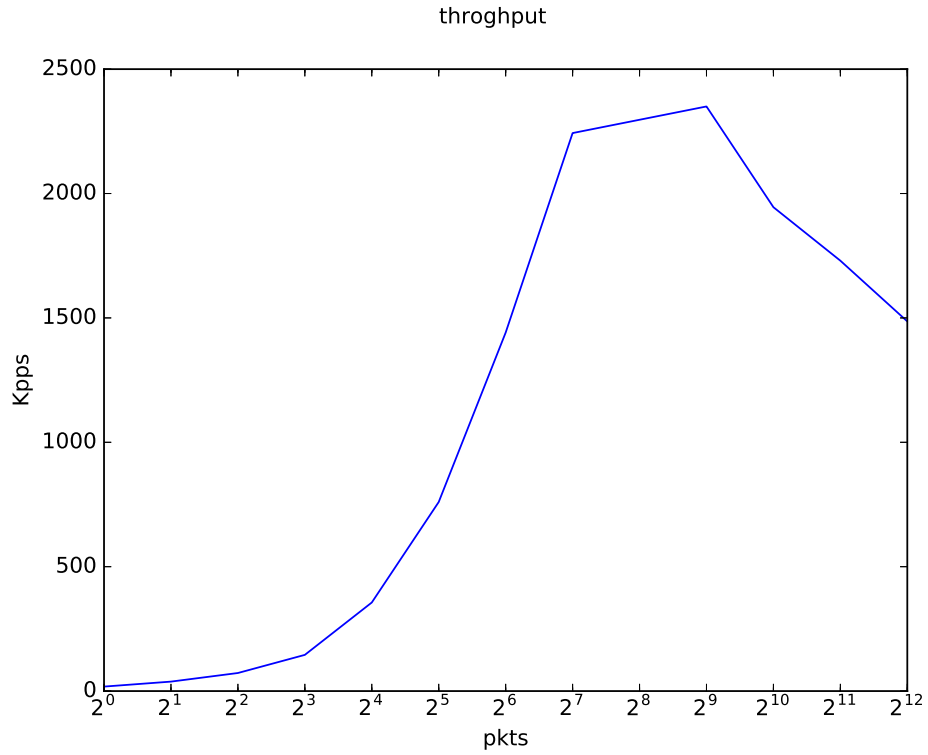


Figure 4.13: throughput vs queue size

From this date we can draw the following conclusions:

- The maximum throughput is achieved with a queue size of 512. This is not surprising since it is exactly the size of a typical netmap batch, so the synchronization costs are reduced because the wait for new packets from the device manager and the queues are in constructive interference.
- The maximum latency does not decrease much varying the queue size. This is because bmv2 is a software switch, and it is implemented in userspace in a non-real-time operating system, and it is very difficult to bound the maximum latency that a packet will experience.

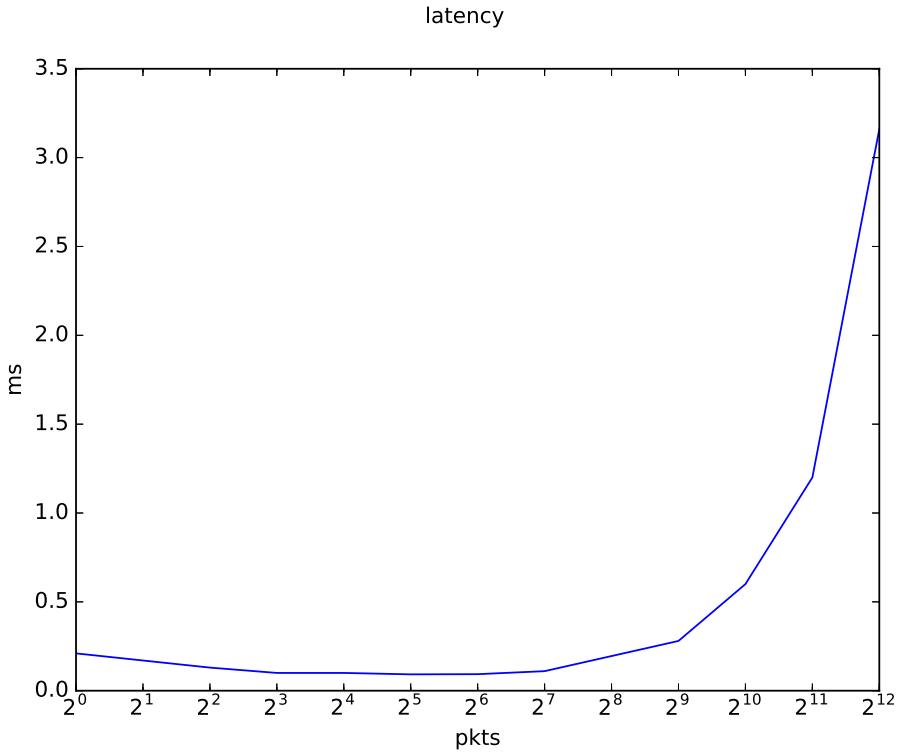


Figure 4.14: average latency vs queue size

# Chapter 5

# Conclusions

In this work we presented an overview of the P4 language and the software implementation of a P4 switch.

We analyzed the performance of the switch and identified the main bottlenecks of the system.

We also optimized some of the bottlenecks, namely the packet I/O, the queuing mechanism, the pipeline structure and memory allocations.

We didn't touch the proper packet processing (parsing, deparsing, ingress and egress controls), mainly because it is pointless to optimize it as long as the overhead costs are a significant portion of the total, and they only increase when the processing costs decrease.

We managed to reduce the overhead cost per packet from 1350 nanoseconds to 200 nanoseconds, for a **6.75x** reduction.

As we can see from fig. 5.1, the extra pipeline stage also reduce the processing time for the simple_router target.

The total speedup for the basic_l2 target is **2.46x**, with a final packet rate of **714Kpps**.

The total speedup for the simple_router target is **3.73x**, with a final packet rate of **2.38Mpps**.

We also managed to fulfill our principles of not breaking existing target code and minimize modifications to the bmv2 codebase, and some patches have already been accepted in the main repository.

## 5.1 Future work

Now that the overhead costs are reduced, further work should focus on improving the processing costs.

The current implementation is fundamentally an interpreter of the P4 language, so a possible way to speed up the processing is to **compile** the P4 program to machine code directly.

The first version of the behavioral model indeed compiled the P4 program in C (and subsequently to machine code), but has been dropped for its lack of dynamic reconfigurability. Possible solutions that keep both the native performance and reconfigurability are:

- P4 program compilation to some kind of efficient bytecode, which then is fed to the software switch.
- Just in Time compilation techniques applied to the current interpreter.

We are confident that with either of these solutions the overall performance could match alternatives SDN solutions like Open vSwitch.

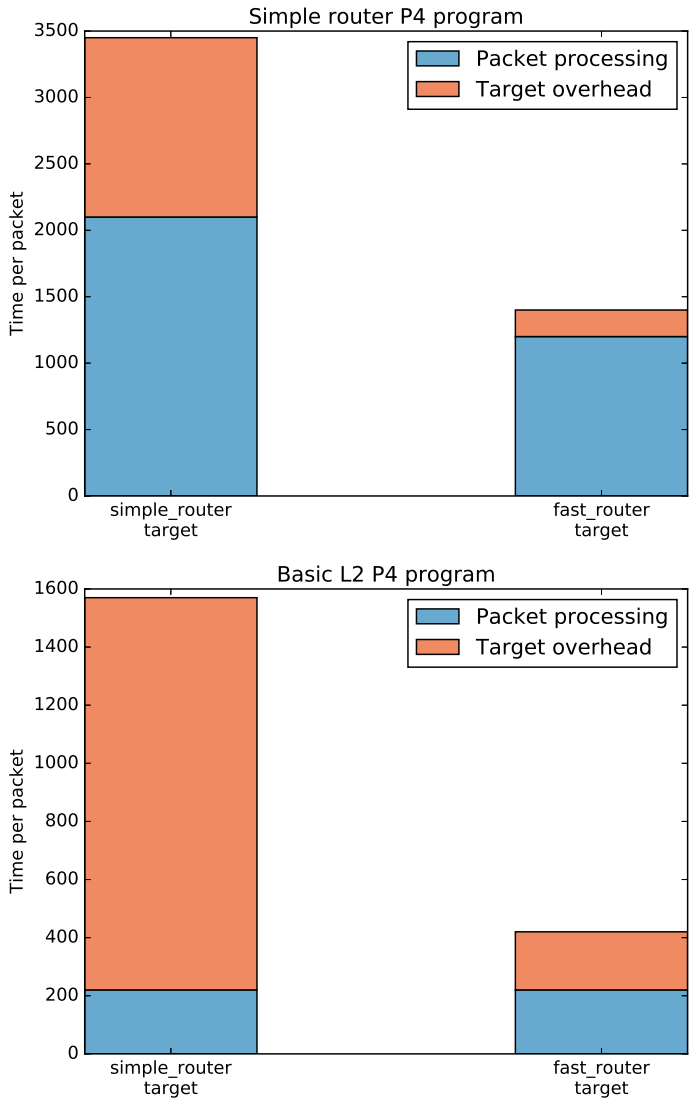


Figure 5.1: Processing and overhead times for the simple_router and fast_switch targets

# Bibliography

[1] "P4 language specification v1.0.2. http://p4.org/wp-content/uploads/2015/04/p4-latest.pdf." 2015.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[3] "Behavioral model repository. https://github.com/p4lang/behavioral-model.".

[4] L. Rizzo, "Netmap: A novel framework for fast packet i/o," in *2012 usenix annual technical conference (usenix atc 12)*, 2012, pp. 101–112.

[5] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[6] "Intel guide for developing multithreaded applications. https://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications." 2012.